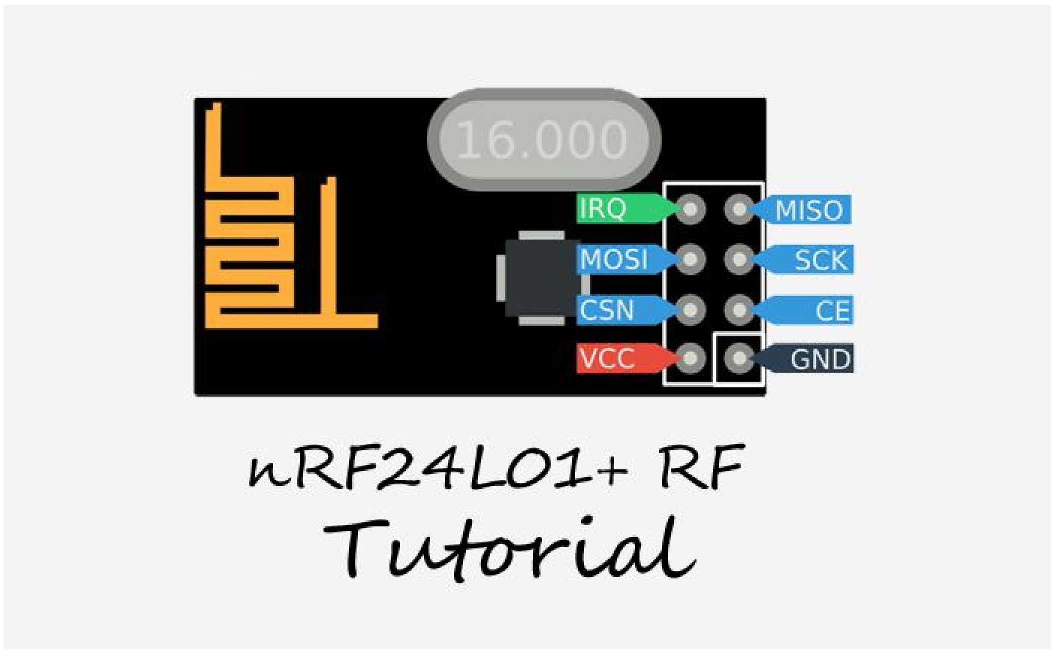
Arduino Beginner's Guide

nRF24L01+ RF Module Tutorial

APRIL 13, 2017 Rahul Iyer



Previously, we covered a tutorial on ESP8266-01 (**ESP8266 Setup Tutorial**), a small-footprint WiFi module designed to allow users to easily add WiFi functionality to their projects. Today, we'll be discussing the **nRF24L01+ RF module**, a kind of sister module to the ESP8266 ESP-01 that allows users to add wireless radio frequency communication to their projects. The nRF24L01+ and the ESP8266 ESP-01 share similar form factors and pin layouts (and even look the same from afar!) but are controlled and function quite differently. In this tutorial, we hope to introduce the fundamentals of using this RF module, while also explaining how it communicates with other RF modules and microcontrollers. For the purposes of this tutorial, we'll be demonstrating interfacing the module with an Arduino Uno microcontroller.

The nRF24L01+ is based on the Nordic Semiconductor nRF24L01+ "RF transceiver IC for the 2.4GHz ISM (Industrial, Scientific, and Medical) band."

Specs:

2.4GHz ISM Band Operation

3.3V Nominal Vcc (5V tolerant inputs)

On-chip Voltage regulation

250kbps, 1 Mbps, 2Mbps on air transmission data rates

Ultra low power operation

Low current draw (900nA – 26μA)

6 data pipes

First, we'll cover the hardware portion of using the module. Similar to the ESP-01, the RF module has a 4 x 2 male header interface. The actual pinout, however, differs from the ESP-01 module because the RF module uses a different communication protocol—SPI—to communicate with other devices. If you'd like to learn more about the SPI protocol, check out our **Arduino Communication Protocols Tutorial**!

The pinout for the RF module is summarized in the following diagram from *Addicore*.

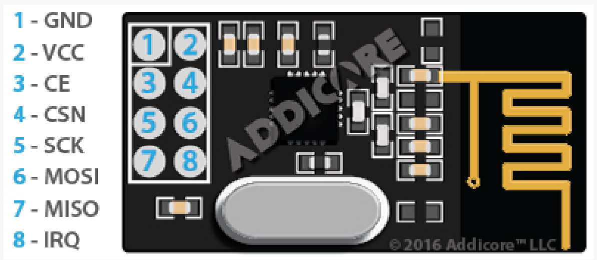


Figure 1. The pinout for the nRF24L01+ RF module. / ©Addicore

The RF module is set up to act as an SPI slave, which means that it can only be used with devices that have dedicated SPI communication lines. This means that the SPI MOSI, MISO, and SCK (clock) pins indicated on the diagram must be connected to their corresponding pins on the microcontroller. On the Arduino, these pins are as follows:

- MOSI: Arduino D11
- MISO: Arduino D12
- SCK: Arduino D13

The CE and the CSN pins can be connected to any output GPIO pin on the Arduino. In software, they are specified appropriately when the SPI communication is initialized.

Here's an example of what the connection between the RF module and the Arduino might look like:

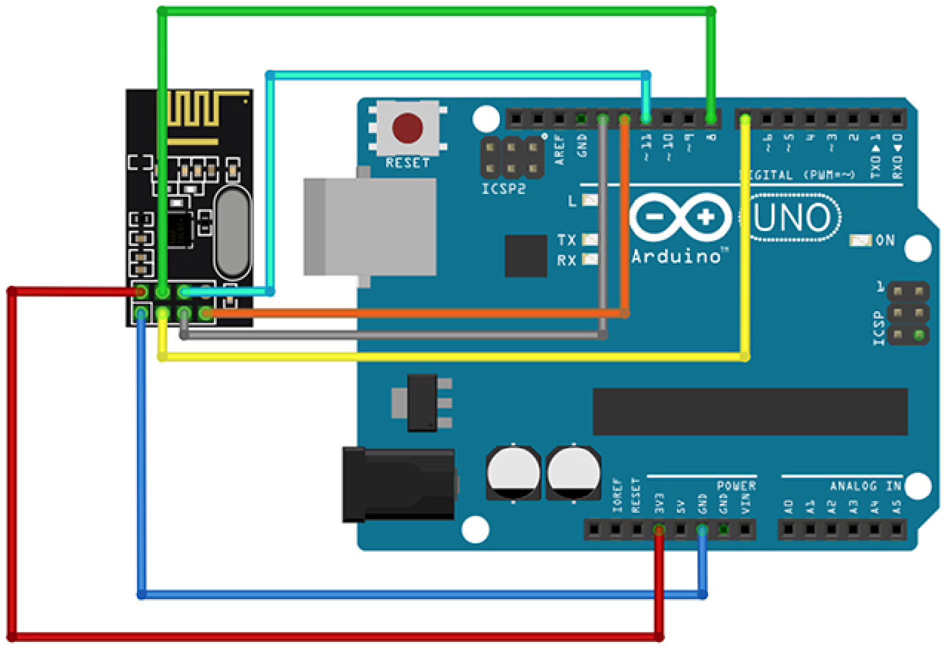


Figure 2. The CE and CSN pins (yellow and green wires in the diagram) can be connected to any two unused digital pins. In the software, you'll specify these pins when constructing the RF module object.

To interface the Arduino with the module, we'll be using TMRh20's RF24 library, which conveniently packages the low-level communications between the RF module and the MCU into an easy-to-use C++ class.

Before we dive into using the module, we'll first cover some fundamentals behind its operation. In the United States, devices that use radio frequency waves are limited to frequency ranges allocated by the FCC. The ISM band is one such range reserved for scientific and medical instruments, and our RF module communicates via frequencies within this ISM range. For the purposes of working with the RF module, it is not necessary to know the details of these frequencies or of how exactly the communication over these frequencies occurs. We'll instead focus on the different aspects of the wireless RF communication that can be controlled.

If you scroll through the *RF24* library documentation, you will notice that there are many parameters that can be set. Some key parameters are

- **Channel:** the specific frequency channel that communication will occur on (frequencies are mapped to integers between 0 and 125
- **Reading pipe:** the reading pipe is a unique 24, 32, or 40-bit address from which the module reads data
- **Writing pipe:** the writing pipe is a unique address to which the module writes data
- **Power Amplifier (PA) level:** the PA level sets the power draw of the chip and thereby the transmission power. For the purposes of this tutorial (use with the Arduino) we will use the minimum power setting.

The RF24 library documentation page provides some great example code for getting started with the library. The example projects can be found here: http://tmrh20.github.io/RF24/examples.html

We'll be taking a look at how the parameters listed above are initialized in the "Getting Started" Arduino code that can be found here:

http://tmrh20.github.io/RF24/GettingStarted_8ino-example.html

GettingStarted.ino

```
/*

* Getting Started example sketch for nRF24L01+ radios

* This is a very basic example of how to send data from one node to another

* Updated: Dec 2014 by TMRh20

*/

#include <SPI.h>

#include "RF24.h"

/****************** User Config ***************************/

/***      Set this radio as radio number 0 or 1         ***/

bool radioNumber = 0;

/* Hardware configuration: Set up nRF24L01 radio on SPI bus plus pins 7 & 8 */

RF24 radio(7,8);

/**********************************************************/

byte addresses[][6] = {"1Node","2Node"};

// Used to control whether this node is sending or receiving

bool role = 0;

void setup() {

Serial.begin(115200);

Serial.println(F("RF24/examples/GettingStarted"));

Serial.println(F("*** PRESS 'T' to begin transmitting to the other node"));

 radio.begin();

// Set the PA Level low to prevent power supply related issues since this is a

// getting_started sketch, and the likelihood of close proximity of the devices. RF24_PA_MAX is default.

radio.setPALevel(RF24_PA_LOW);

 // Open a writing and reading pipe on each radio, with opposite addresses

```

```
if(radioNumber){

  radio.openWritingPipe(addresses[1]);

  radio.openReadingPipe(1,addresses[0]);

}else{

  radio.openWritingPipe(addresses[0]);

  radio.openReadingPipe(1,addresses[1]);

}

 // Start the radio listening for data

radio.startListening();

}

void loop() {

/****************** Ping Out Role ***************************/

if (role == 1)  {



  radio.stopListening();                                    // First, stop listening so we can talk.







  Serial.println(F("Now sending"));

  unsigned long start_time = micros();                             // Take the time, and send it.  This will block until

   if (!radio.write( &start_time, sizeof(unsigned long) )){

     Serial.println(F("failed"));

   }



  radio.startListening();                                    // Now, continue listening



  unsigned long started_waiting_at = micros();               // Set up a timeout period, get the current microseconds

  boolean timeout = false;                                   // Set up a variable to indicate if a response was received



  while ( ! radio.available() ){                             // While nothing is received

    if (micros() - started_waiting_at > 200000 ){            // If waited longer than 200ms, indicate timeout and exit wh

        timeout = true;

        break;

    }

  }




  if ( timeout ){                                             // Describe the results

      Serial.println(F("Failed, response timed out."));

  }else{

      unsigned long got_time;                                 // Grab the response, compare, and send to debugging spew

      radio.read( &got_time, sizeof(unsigned long) );

      unsigned long end_time = micros();



      // Spew it

      Serial.print(F("Sent "));

      Serial.print(start_time);

      Serial.print(F(", Got response "));

      Serial.print(got_time);

      Serial.print(F(", Round-trip delay "));

      Serial.print(end_time-start_time);

      Serial.println(F(" microseconds"));

  }

  // Try again 1s later

  delay(1000);

}

/****************** Pong Back Role ***************************/

if ( role == 0 )

{

  unsigned long got_time;



  if( radio.available()){

                                                                  // Variable for the received timestamp

    while (radio.available()) {                                   // While there is data ready

      radio.read( &got_time, sizeof(unsigned long) );             // Get the payload

    }



    radio.stopListening();                                        // First, stop listening so we can talk

    radio.write( &got_time, sizeof(unsigned long) );              // Send the final one back.

    radio.startListening();                                       // Now, resume listening so we catch the next packets.

    Serial.print(F("Sent response "));

    Serial.println(got_time);

 }

}

/****************** Change Roles via Serial Commands ***************************/

if ( Serial.available() )

{

  char c = toupper(Serial.read());

  if ( c == 'T' && role == 0 ){

    Serial.println(F("*** CHANGING TO TRANSMIT ROLE -- PRESS 'R' TO SWITCH BACK"));

    role = 1;                  // Become the primary transmitter (ping out)



 }else

  if ( c == 'R' && role == 1 ){

    Serial.println(F("*** CHANGING TO RECEIVE ROLE -- PRESS 'T' TO SWITCH BACK"));

     role = 0;                // Become the primary receiver (pong back)

     radio.startListening();



  }

}

} // Loop
```

The first two lines of interest are the two C++ #include directives at the top of the file: one for including the Arduino SPI library (recall from earlier that the RF module uses SPI to communicate with the Arduino), and one for including the RF24 library.

```
#include <SPI.h>
#include “RF24.h”
```

The next line of interest is the line that constructs the RF24 object: RF24 radio(7,8); The two parameters passed to the constructor are the CE and CSN digital pins that are connected to the module. Although the MOSI, MISO, and SCK pins must be digital pins 11, 12, and 13 respectively, the CE and CSN pins can be *any* two digital pins!

Next we'll look at the pipe addresses for reading and writing. As you can probably guess, the writing and reading pipe addresses are swapped between the two radios that are communicating with each other, as the writing pipe for each radio is the reading pipe for the other. The radio addresses are of size 24, 32, or 40 bit. In the example code, these addresses are converted from C++ string literals, but you can also specify them in binary or hexadecimal format. For example, a 40-bit address specified as a hex value might be 0xF0F0F0F0F0. A good programming practice for storing the writing and reading pipe addresses is to put the two values inside an array. In the example code, the writing and reading pipe addresses are stored in a byte array named "addresses."

In the *void setup()* method, we need to provide instructions for initializing the radios with the address pipe parameters and the other parameters as well.

First, the *RF24::begin()* method is called. The *begin()* must be called before any of the other RF24 library methods are called on the *radio* object because it initializes the operation of the RF chip.

Next, the power amplifier (PA) level is set by calling the *RF24::setPALevel()* method. The RF24 library provides different constant values to specify the power amplifier level. Higher PA levels mean that the module can communicate over longer distances, but in turn draws more current during operation. For use in the getting started sketch, we pass the ***RF_24_LOW*** constant as a parameter to the *setPALevel()* method because the distance between the two communicating modules is not going to be very great. In general, when using the RF module with an Arduino board, it is probably a good idea to keep the PA level as low as possible to reduce the current draw on the Arduino's regulated power supply.

Next, we'll look at how to initialize the writing and reading pipes. We've already defined the writing and reading pipes as some byte values. We must now pass these definitions to the ***radio*** object so that it also has knowledge of the writing and reading pipe addresses. The writing pipe is set with the *openWritingPipe()* method, and the reading pipe is set with the *openReadingPipe()* method. An example of opening a writing and reading pipe is:

radio.openWritingPipe(addresses[1]);

radio.openReadingPipe(1, addresses[0]);

Note that the *openReadingPipe()* method must be passed an additional integer parameter that describes which reading pipe is being initialized. This is because the RF module can have up to 6 reading pipes open at a given time!

The example code appropriately assigns the reading and writing pipe values through a *role* boolean value. Based on the value of *role*, the program determines whether the RF module is the *ping* or *pong* device. Similar code can be employed in your projects as well. Just ensure that the reading and writing addresses are swapped on the two devices, or no data will be transmitted or read!

The radio module is instructed to begin listening by calling the *RF24::startListening()* method. It is important to note that the reading pipe must be initialized before the RF module is instructed to begin listening for data (i.e. the *openReadingPipe()* method must be called before the *startListening()* method!)

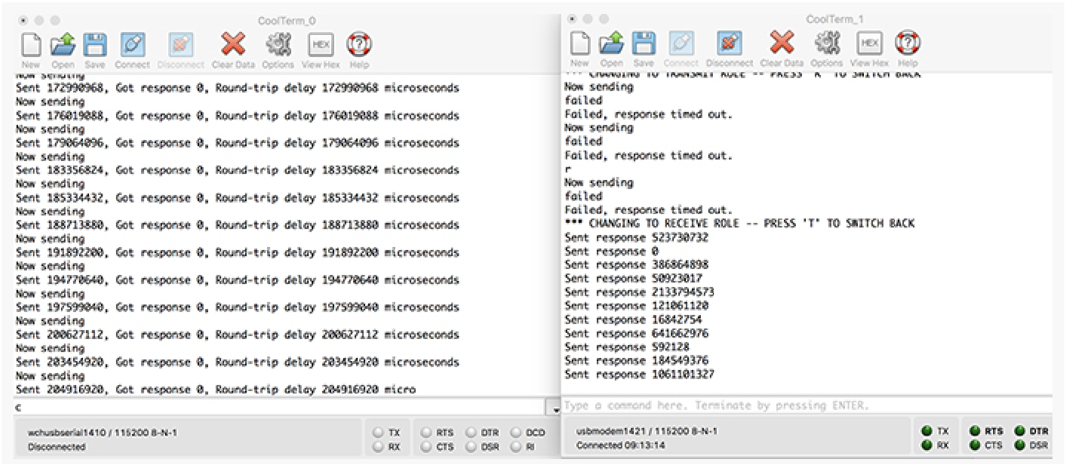
Similarly, the RF24 class also provides a *stopListening()* method, which must be called before the radio module can begin writing.

In the example code, you may notice that the radio module is instructed to check for incoming data through use of the *RF24::available()* method. This is similar to the *Serial::available()* and *SoftwareSerial::available()* methods that we've seen before–if data is available over the RF connection, the *available()* method returns true, and the data can be read.

Finally, the RF24 class provides methods to actually write and read data. The parameters for the *RF24::write()* and *RF24::read()* methods are (1) a pointer to a variable of the same type as the data being transmitted, and (2) the size of the data being transmitted. In the *read()* method, the variable to which the pointer points receives the data being read. In the *write()* method, the variable to which the pointer points holds the data that is being written. In both methods, it is absolutely necessary to ensure that the pointers point to variables of the same type as the data being transmitted, and that the size passed to the method actually reflects the size of the data. Passing incorrect types or values of size to the *read()* and *write()* methods can result in undesired value truncations, which may render the data transmitted useless. In the "Getting Started" code, the data being transmitted is an *unsigned long*. Therefore the pointer passed as an argument to the *read()* and *write()* methods points to a variable also of type *unsigned long*. It is also clear that the size of the data transmitted is always the size of an unsigned long. In this situation, the size does not need to be passed explicitly as an integer, and the size argument can instead simply be ***sizeof(unsigned long)***.

The only parameter that the "Getting Started" code does not cover is the communication channel. If a specific channel is desired (for example, if you have multiple RF networks that you do not want to interfere with each other), then the channel can be set by passing an 8-bit integer parameter to the *RF24::setChannel()* method. An example of this code might be *radio.setChannel(10)*;

Try connecting two RF modules to two separate Arduino boards and upload the "Getting Started" code on each (for one of the boards, you will have to change the *role* boolean to 1). You should be now be able to send and receive back messages with corresponding ping times! Here's an image of the two "ping" and "pong" serial monitors side by side:



```
Sent 172990968, Got response 0, Round-trip delay 172990968 microseconds
Now sending
Sent 176019088, Got response 0, Round-trip delay 176019088 microseconds
Now sending
Sent 179064096, Got response 0, Round-trip delay 179064096 microseconds
Now sending
Sent 183356824, Got response 0, Round-trip delay 183356824 microseconds
Now sending
Sent 185334432, Got response 0, Round-trip delay 185334432 microseconds
Now sending
Sent 188713880, Got response 0, Round-trip delay 188713880 microseconds
Now sending
Sent 191892200, Got response 0, Round-trip delay 191892200 microseconds
Now sending
Sent 194770640, Got response 0, Round-trip delay 194770640 microseconds
Now sending
Sent 197599040, Got response 0, Round-trip delay 197599040 microseconds
Now sending
Sent 200627112, Got response 0, Round-trip delay 200627112 microseconds
Now sending
Sent 203454920, Got response 0, Round-trip delay 203454920 microseconds
Now sending
Sent 204916920, Got response 0, Round-trip delay 204916920 micro
c
```

wchusbserial1410 / 115200 8-N-1
Disconnected
TX RX RTS CTS DTR DSR DCD RI

```
Now sending
failed
Failed, response timed out.
Now sending
failed
Failed, response timed out.
r
Now sending
failed
Failed, response timed out.
*** CHANGING TO RECEIVE ROLE -- PRESS 'T' TO SWITCH BACK
Sent response 523730732
Sent response 0
Sent response 386864898
Sent response 50923017
Sent response 2133794573
Sent response 121061120
Sent response 16842754
Sent response 641662976
Sent response 592128
Sent response 184549376
Sent response 1061101327
```

Type a command here. Terminate by pressing ENTER.

usbmodem1421 / 115200 8-N-1
Connected 09:13:14
TX RX RTS CTS DTR DSR

Figure 3. Left: "ping" monitor statements (device 1) ; Right "pong" monitor statements (device 0)

Congratulations on making it through the nRF24L01+ tutorial! You are now equipped with the skills and knowledge to make your own projects using these nifty RF modules! You can check out the Device Plus blog for projects that involve these RF modules!

0 Comments **www.deviceplus.com** 1 **Login**

Recommend Share Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

ALSO ON **WWW.DEVICEPLUS.COM**

ESP8266 Setup Tutorial using Arduino

2 comments • 5 months ago

Peterson de Aquino — Hi thanks for you post! Can you send me more informations about ESP8266? Exemplos or tutorial because my project don´t …

JPEG Decoding on Arduino Tutorial

45 comments • 5 months ago

Allan T. — Sad to hear that my friend, but it's okay. I'll check you links above, and see if I can get some clues on what I'm trying to do. Thanks for …

Arduino-Based DIY Electric Skateboard

5 comments • 5 months ago

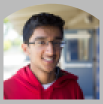
Emmanuel Nieves — A motor with more Watts will only give your skateboard more torque.If you want more speed you will need a combination of …

Make Your Own Arduino RFID Door Lock

2 comments • 5 months ago

Tiberia Todeilă — Thanks for the suggestion!

Subscribe Add Disqus to your siteAdd DisqusAdd Privacy DISQUS



Rahul Iyer

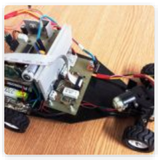
Studying Electrical Engineering at UCLA, Rahul loves to work on electronics and robotics projects as a hobby. He is especially enthusiastic about electric vehicle technology and assistive robotics.

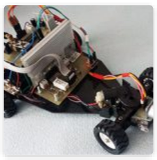
Check us out on Social Media

Facebook | Twitter | Linkedin

Reddit | Google+ | Follow on Feedly

Recommended Posts


Arduino Robot RF Explorer – Mechanics – Part 1


Arduino Robot RF Explorer – Part 2 – Putting Everything Together

Top 6 Energy Harvesting Technologies from IDTechEx USA 2016 (Cont.)

ESP8266 Setup Tutorial using Arduino

Intro to FPGAs with the Mojo – Part 2

ESP-WROOM-02 Wifi Setup Guide

Receive update on new posts
Privacy Policy

Submit

© COPYRIGHT 2016. DEVICE PLUS - POWERED BY ROHM